Глава 4

# Нисходящий синтаксический анализ

## 4.1. Вступление

В данной главе будут рассмотрены принципы нисходящего синтаксического анализа, а также их применение на практике. Методы нисходящего синтаксического анализа являются более интуитивными, чем методы восходящего анализа. Поэтому вначале будет рассмотрен первый из указанных случаев. Впрочем, для восходящего синтаксического анализа разработано больше инструментальных средств, и он может применяться более широко, чем нисходящий анализ. Подробно методы восходящего синтаксического анализа будут описаны в главе 5.

В настоящей главе будут рассмотрены следующие вопросы.

- Критерии принятия решений, применяемые при нисходящем синтаксическом анализе.
- Контекстно-свободные грамматики, на которых может основываться нисходящий синтаксический анализ.
- Простые методы создания нисходящих синтаксических анализаторов с использованием соответствующих грамматик.
- Преобразование грамматики в форму, подходящую для нисходящего анализа.
- Преимущества и недостатки нисходящего синтаксического анализа.
- Использование контекстно-свободных грамматик как основы процессов времени компиляции.

## 4.2. Критерии принятия решений

Напомним, что задача синтаксического анализа состоит в нахождении порождения (если таковое существует) конкретного выражения с использованием данной грамматики. При нисходящем анализе в большинстве случаев требуется найти *левое порождение*. При обратном порядке разбора чаще всего искомым

является *правое порождение*. Следует помнить, что при нисходящем синтаксическом анализе мы начинаем с символа предложения и *генерируем* предложение, тогда как при восходящем анализе имеется предложение, которое *сворачивается* в символ предложения. Далее будем предполагать, что предложения, которые предстоит сгенерировать или свернуть, читаются слева направо (хотя теоретически возможны и обратные проходы, когда предложения читаются справа налево).

В разделе 2.5 был рассмотрен язык

$\{x^m y^n \mid m, n > 0\}$,

сгенерированный следующими продукциями.

$S \rightarrow XY$
$X \rightarrow xX$
$X \rightarrow x$
$Y \rightarrow yY$
$Y \rightarrow y$

Было показано, что предложение

$xxxyy$

можно породить (или сгенерировать) с помощью левого порождения, а именно:

$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxxY \Rightarrow xxxyY \Rightarrow xxxyy$

Первый шаг порождения очевиден, поскольку символ предложения $S$ находится с левой стороны только одной продукции.

$S \Rightarrow XY$

Следующий шаг несколько сложнее, поскольку крайний левый нетерминал в сентенциальной форме ($X$) входит в левую часть более одной продукции (в данном случае — в две продукции). В то же время следует помнить, что при синтаксическом анализе конечный результат (сгенерированное предложение) всегда известен. В данном случае результат будет следующим.

$xxxyy$

Поскольку в указанном выражении более одного $x$, следующей используемой продукцией должна быть такая.

$X \rightarrow xX$

Подобным образом на третьем шаге порождения должна использоваться та же продукция. В результате получим следующее.

$xxXY$

Четвертый шаг имеет вид

$xxXY \Rightarrow xxxY$

Видим, что, поскольку требуется сгенерировать последний $x$, в первый раз используется продукция

$X \rightarrow x$

На пятом шаге

$$xxxY \Rightarrow xxxyY$$

используется следующая продукция.

$$Y \rightarrow yY$$

Использование именно этой продукции объясняется тем, что далее также требуется сгенерировать *y*. Поскольку в дальнейшем *уже не требуется* генерировать *y*, то последний шаг порождения

$$xxxyY \Rightarrow xxxyy$$

подразумевает использование следующей продукции.

$$Y \rightarrow y$$

В приведенном выше примере найти порождение было нетрудно, так как было известно выражение, которое следовало сгенерировать, хотя на большинстве этапов было необходимо знать *два* символа помимо уже сгенерированных. При использовании некоторых грамматик для нахождения правильного порождения требуется более двух символов (в некоторых случаях это число может быть произвольным). В дальнейшем нас интересуют такие грамматики, которым для определения правильного порождения требуется не более одного *символа предпросмотра* на каждом этапе порождения.

Еще один наглядный пример, демонстрирующий этапы нисходящего порождения, приводится в табл. 4.1. Знаки предложения рассматриваются по одному и используются для управления процессом синтаксического анализа. После генерации знак предложения зачеркивается (в столбце "входная строка"). Каждому этапу синтаксического анализа соответствуют три позиции таблицы: входная строка с вычеркнутыми символами, текущая продукция, а также текущее состояние сентенциальной формы. В конце синтаксического анализа все знаки входной строки зачеркнуты, а сентенциальная форма соответствует исходной заданной строке.

***Таблица 4.1.***

| *Входная строка* | *Продукция* | *Сентенциальная форма* |
|---|---|---|
| *xxyy* | $S \rightarrow XY$ | *XY* |
| *xxyy* | $X \rightarrow xX$ | *xXY* |
| ~~*x*~~*xyy* | $X \rightarrow xX$ | *xxXY* |
| ~~*xx*~~*yy* | $X \rightarrow x$ | *xxxY* |
| ~~*xxx*~~*yy* | $Y \rightarrow yY$ | *xxxyY* |
| ~~*Xxxy*~~*y* | $Y \rightarrow y$ | *xxxyy* |
| ~~*xxxy*~~*y* | | |

На каждом этапе первый незачеркнутый символ входной строки определяется как *входной символ* и используется для разбора. Если в сентенциальной форме генерируется терминал, появляется еще один зачеркнутый символ. По

определению, *символ предпросмотра* (lookahead symbol) — это либо текущий входной символ, либо маркер конца (специальный символ, стоящий в конце строки; обычно обозначается как ⊥). Принятие решений при нисходящем синтаксическом разборе, как правило, основывается на символе (или последовательности символов) предпросмотра. Кроме того, существуют более общие методы, в которых учитывается история синтаксического анализа.

## 4.3. LL(1)-грамматики

В данном разделе рассматриваются свойства грамматик, поддерживающих методы нисходящего синтаксического анализа с одним символом предпросмотра. Будем считать, что грамматики являются однозначными, так что каждому предложению языка соответствует единственное левое порождение. В данном случае для каждого нетерминала, который находится в левой части нескольких продукций, необходимо найти такие непересекающиеся множества символов предпросмотра, чтобы каждое множество содержало символы, соответствующие точно одной возможной правой части. Выбор конкретной продукции для замены данного нетерминала будет определяться символом предпросмотра и множеством, к которому принадлежит данный символ. Объединение различных непересекающихся множеств для заданного нетерминала не обязательно должно составлять алфавит, на котором определен язык. Если символ предпросмотра не принадлежит ни одному из непересекающихся множеств, можно сделать вывод о наличии синтаксической ошибки.

Множество символов предпросмотра, соотнесенных с применением определенной продукции, называется ее *множеством первых порождаемых символов* (director symbol set). Перед определением данного понятия, определим вначале следующие два.

1. *Стартовый символ* данного нетерминала определяется как любой символ (например, терминал), который может появиться в начале строки, генерируемой нетерминалом.
2. *Символ-последователь* данного нетерминала определяется как любой символ (терминал или нетерминал), который может следовать за нетерминалом в любой сентенциальной форме.

Вычисление множества стартовых символов может быть достаточно трудоемким и вычислительно сложным процессом, и оно всегда выполняется в процессе генерации программы синтаксического анализа, а не при каждом запуске этой программы. Впрочем, возможны ситуации, когда упомянутые вычисления относительно просты. Пусть, например, (используется принятая ранее договоренность об обозначении терминалов и нетерминалов) для нетерминала $T$ грамматика содержит только две продукции.

$T \rightarrow aG$
$T \rightarrow bG$

В этом случае имеем следующие множества стартовых символов.

| *Продукция* | *Множество стартовых символов* |
|---|---|
| $T \to aG$ | $\{a\}$ |
| $T \to bG$ | $\{b\}$ |

В общем случае, если продукция начинается с терминала, ее множество стартовых символов просто состоит из этого терминала. В то же время, если продукция не начинается с терминала, для нее все равно нужно вычислить множество стартовых символов. Пусть в рассматриваемой грамматике имеются следующие продукции для нетерминала $R$.

$R \to BG$
$R \to CH$

Тогда множество стартовых символов для этих продукций нельзя определить "с ходу". В то же время, пусть имеются только следующие продукции для нетерминала $B$.

$B \to cD$
$B \to TV$

Тогда можно заключить, что множеством стартовых символов для продукции

$R \to BG$

будет набор

$\{a, b, c\}$,

состоящий из всех стартовых символов для $B$.

Введение символа $c$ в множество стартовых является очевидным (см. первую продукцию для $B$), а введение набора $\{a, b\}$ объясняется тем, что эти символы являются стартовыми для $T$. В общем случае ситуация может быть значительно сложнее. Пусть имеется следующая последовательность продукций.

$A \to BC$; $B \to DE$; $D \to FG$; $F \to HI$; $H \to xY$

Из данных продукций следует, в частности, что $x$ является стартовым символом для продукции

$A \to BC$

Еще одним источником сложностей можно назвать нетерминалы, которые могут генерировать пустые строки. Допустим, имеются следующие продукции.

$A \to BC$
$B \to \varepsilon$

В этом случае множество стартовых символов для продукции $A \to BC$ будет включать стартовые символы $C$, а также стартовые символы $B$ (определяемые не приведенными здесь продукциями). Если оба нетерминала $B$ и $C$ могут генерировать пустые строки, то на использование данной продукции будут указывать символы предпросмотра, являющиеся последователями $A$ и стартовыми символами $BC$. *Множество первых порождаемых символов* продукции выби-

рается как множество *всех* терминалов, которые, выступая как символы предпросмотра, указывают на использование данной продукции. Таким образом, множество первых порождаемых символов для продукции

$A \rightarrow BC$

будет включать все *символы-последователи* $A$, а также стартовые символы $BC$.

Рассмотрим грамматику со следующими продукциями.

$S \rightarrow Ty$
$T \rightarrow AB$
$T \rightarrow sT$
$A \rightarrow aA$
$A \rightarrow \varepsilon$
$B \rightarrow bB$
$B \rightarrow \varepsilon$

Ниже приводятся множества первых порождаемых символов для различных продукций данной грамматики.

| *Продукция* | *Множество первых порождаемых символов* |
|---|---|
| $T \rightarrow AB$ | $\{a, b, y\}$ |
| $T \rightarrow sT$ | $\{s\}$ |
| $A \rightarrow aA$ | $\{a\}$ |
| $A \rightarrow \varepsilon$ | $\{b, y\}$ |
| $B \rightarrow bB$ | $\{b\}$ |
| $B \rightarrow \varepsilon$ | $\{y\}$ |

Множество первых порождаемых символов для продукции $B \rightarrow \varepsilon$ равно $\{y\}$, поскольку $y$ может следовать за $B$; подобным образом, множество первых порождаемых символов для продукции $A \rightarrow \varepsilon$ равно $\{b, y\}$, поскольку $b$ и $y$ могут следовать за $A$, если $B$ генерирует пустую строку.

Существует алгоритм (см. раздел дополнительной литературы в конце главы) поиска множеств первых порождаемых символов для всех продукций грамматики. Сложность этого алгоритма, в основном, связана с тем, что символы могут генерировать пустые строки; в данной книге этот алгоритм приводиться не будет. После вычисления всех множеств первых порождаемых символов их можно проверить на предмет пересечения. *LL(1)-грамматику* можно определить как грамматику, в которой для каждого нетерминала, появляющегося в левой части нескольких продукций, множества первых порождаемых символов всех продукций, в которых появляется этот нетерминал, являются непересекающимися. Термин LL(1) имеет следующее происхождение: первое L означает чтение слева (*L*eft) направо, второе L означает использование левых (*L*eftmost) порождений, а 1 — один символ предпросмотра.

Описанная выше грамматика, очевидно, является LL(1)-грамматикой, поскольку множества символов предпросмотра для $T$, $A$ и $B$ не пересекаются. LL(1)-грамматики формируют основу методов нисходящего анализа, описыва-

емых в данной главе. Если вычислены все множества первых порождаемых символов для всех возможных правых частей продукций, то языки, которые описываются LL(1)-грамматикой, всегда анализируются детерминировано, т.е. без необходимости отменять продукцию после ее применения. Существуют более распространенные классы грамматик, которые могут использоваться для детерминированного нисходящего анализа, но обычно используются именно LL(1)-грамматики. Недетерминированный нисходящий анализ, основанный на откате (backtracking), уже не считается эффективной процедурой, хотя в начале эры компиляторов он широко использовался в языках, подобных FORTRAN. Грамматики LL($k$), требующие $k$ символов предпросмотра для различения альтернативных правых частей, также уже не считаются практичными с точки зрения синтаксического анализа.

*LL(1)-язык* — это язык, который можно сгенерировать посредством LL(1)-грамматики. Отсюда следует, что для любого LL(1)-языка возможен нисходящий синтаксический анализ с одним символом предпросмотра. Рассмотрим некоторые "теоретические" результаты, связанные с LL(1)-грамматиками и языками, после чего перейдем к более практическим вопросам реализации.

Во-первых, как было сказано выше, существует алгоритм определения, относится ли данная грамматика к классу LL(1), поэтому грамматику можно проверить на "LL(1)-ность" прежде, чем создавать на ее основе программу синтаксического анализа. В то же время, что может несколько удивить на первый взгляд, не существует алгоритма определения, относится ли данный язык к классу LL(1), т.е. имеет он LL(1)-грамматику или нет. Это означает, что не-LL(1)-грамматика может иметь или не иметь эквивалентную LL(1), генерирующую тот же язык, и не существует алгоритма, который для данной произвольной грамматики определит, является ли генерируемый ею *язык* LL(1) или нет. Разумеется, существуют алгоритмы, которые могут использоваться для частных случаев, например, если грамматика является LL(1) — язык также является LL(1); также можно выделить определенные классы грамматик, которые никогда не будут генерировать LL(1)-языки. В то же время, в общем случае задача является неразрешимой в том же смысле, как неразрешимы задача определения однозначности языка и проблема остановки для машин Тьюринга.

Приведенный выше результат является важным, поскольку далее будет показано, что имеются грамматики, не являющиеся LL(1), которые, тем не менее, генерируют LL(1)-языки, т.е. грамматики имеют эквивалентные LL(1)-грамматики. Это означает, что грамматики часто нужно преобразовывать, прежде чем использовать с методами нисходящего синтаксического анализа. Фактически, грамматики, которые обычно используются в определениях языков или в учебниках, редко являются LL(1) и, следовательно, не могут непосредственно использоваться для эффективного нисходящего анализа. Тем больше оснований сожалеть, что не существует алгоритма определения, имеет ли грамматика эквивалентную LL(1), а это означает, что *в любом случае* не существует алгоритма поиска эквивалентной LL(1)-грамматики, если даже такая

грамматика и существует. Впрочем, в этом несовершенном мире иногда приходится иметь дело с несовершенными алгоритмами, так что, если нет алгоритма выполнения преобразования в общем случае (т.е. для всех случаев), имеются алгоритмы, работающие для большого числа групп частных случаев, которые *никогда* не дают неверного результата. В то же время эти алгоритмы иногда могут зацикливаться.

Не стоит пугаться, что существует одно свойство грамматики, которое (если оно присутствует) препятствует тому, чтобы грамматика была LL(1), и это — левая рекурсия. Рассмотрим следующие продукции.

$D \rightarrow Dx$
$D \rightarrow y$

Обозначив через *DS* множество первых порождаемых символов (director symbol set), можем записать следующее:

$DS(D \rightarrow Dx) = \{y\}$
$DS(D \rightarrow y) = \{y\}$

Здесь второе множество первых порождаемых символов следует непосредственно из продукции, а первое следует из того, что *D* является стартовым символом правой части. Очевидно, что никакая грамматика, имеющая подобную левую рекурсию, не может быть LL(1). Предположим, впрочем, что для *D* имеется *единственная* продукция.

$D \rightarrow Dx$

В этом случае, разумеется, в алгоритме по определению принадлежности к классу LL(1) вообще не будут найдены первые порождаемые символы для *D*. Использование данной продукции никогда не даст ни одной строки терминалов, поскольку не существует способа избавиться от нетерминала *D*, если таковой появится в сентенциальной форме. Грамматика с продукциями, которые не могут использоваться или (по каким-то причинам) не являются необходимыми, часто называется *нечистой* (unclean); далее предполагается, что все рассматриваемые грамматики являются “чистыми”.

Левая рекурсия может быть непрямой, включающей две или более продукции. Рассмотрим, например, следующий набор продукций.

$A \rightarrow BC$
$B \rightarrow DE$
$D \rightarrow FG$
$F \rightarrow AH$

Здесь имеет место непрямая левая рекурсия, в которой задействованы нетерминалы *A*, *B*, *D* и *F*. Разумеется, должны существовать нерекурсивные правила, по крайней мере, для некоторых нетерминалов, гарантирующие чистоту грамматики. Как и для прямой левой рекурсии, любая грамматика, имеющая непрямую левую рекурсию, будет характеризоваться пересекающимися множествами первых порождаемых символов для некоторых нетерминалов и, следовательно, такая грам-

матика не может быть LL(1). Итак, ни одна леворекурсивная грамматика не является LL(1). Это не представляет такой уж серьезной проблемы, как может показаться на первый взгляд, поскольку можно показать (см. раздел 4.5), что *все* левые рекурсии грамматики можно заменить правыми, не затронув генерируемый язык.

Преобразования грамматик, выполняемые автоматически или вручную, являются неотъемлемой частью LL(1)-анализа и, как говорят многие, одним из его ограничений. Впрочем, при наличии соответствующей грамматики написание программы синтаксического анализа является простой задачей. Кто-то может даже сказать, что написать эту программу можно с той же скоростью, с которой вы набираете на клавиатуре, используя при этом известный метод *рекурсивного спуска*, который будет описан в следующем ниже разделе.

# 4.4. Рекурсивный спуск

Синтаксический анализ методом *рекурсивного спуска* (recursive descent) включает использование рекурсивных процедур и работает нисходящим образом — отсюда и название! Предположим, например, что в грамматике языка программирования имеется символ предложения `PROGRAM` и единственное правило с символом предложения с левой стороны.

*PROGRAM* → *begin DECLIST comma STATELIST end*

Как обычно, слова из прописных букв представляют нетерминальные символы, а слова из строчных букв — терминальные символы. Использование синтаксического анализа методом рекурсивного спуска состоит из последовательного определения всех символов правой части. Появление слова ***begin*** определяется посредством вызова лексического анализатора, *DECLIST* определяется вызовом функции (для удобства названной `DECLIST`), ***comma*** определяется непосредственно, с помощью лексического анализатора, *STATELIST* определяется посредством вызова функции, именуемой `STATELIST`, наконец, `end` определяется непосредственно, снова с помощью лексического анализатора.

Пусть другие продукции грамматики имеют следующий вид.

*DECLIST* → *d semi DECLIST*
*d*
*STATELIST* → *s semi STATELIST*
*s*

Здесь ***d*** можно рассматривать как объявление (declaration), а ***s*** — как оператор (statement), но на данный момент оба символа считаются просто терминалами.

Метод рекурсивного спуска может применяться только к LL(1)-грамматикам, но данная грамматика, очевидно, такой не является, поскольку

*DS*(*DECLIST* → *d semi DECLIST*) = {*d*}
*DS*(*DECLIST* → *d*) = {*d*}

Данные множества не являются непересекающимися; то же можно сказать и для продукций с нетерминалом *STATELIST*. Итак, данную грамматику требуется пре-

образовать. Для выполнения необходимого преобразования продукций для нетерминала *DECLIST* вначале следует отметить, что данные две продукции генерируют последовательности следующего вида:

*d*
*d semi d*
*d semi d semi d*

и т.д., причем последний терминал *d* генерируется вторым правилом для нетерминала *DECLIST*, а остальные — первым. Полное множество подобных последовательностей можно записать как регулярное выражение.

$d(semi\ d)^*$

Каждое предложение можно рассматривать как начинающееся с *d*, за которым следует либо *пустая строка*, либо символ *semi*, за которым идет что угодно, составляющее *DECLIST*. Следовательно, две продукции можем переписать в следующем виде.

$DECLIST \rightarrow dX$
$X \rightarrow semi\ DECLIST$
$\varepsilon$

Итак, в грамматике появился новый нетерминал *X*.

Подобным образом можно переписать продукции для *STATELIST*.

$STATELIST \rightarrow sY$
$Y \rightarrow semi\ DECLIST$
$\varepsilon$

Получили еще один новый нетерминал — *Y*.

Преобразования грамматики не являются совсем очевидными. Простейшим путем их определения является рассмотрение *языка*, генерируемого продукциями, подлежащими преобразованию, как это было сделано выше. В то же время данный тип преобразования является настолько общим, что его легко определить и выполнить вручную или автоматически. Данный процесс часто называется *факторизацией*, по аналогии с соответствующим алгебраическим процессом. Фактически, грамматику полезно рассматривать как некоторую алгебру с присущими ей правилами преобразования, не затрагивающими язык в целом.

Перечислим продукции преобразованной грамматики.

$PROGRAM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$
$DECLIST \rightarrow dX$
$X \rightarrow semi\ DECLIST$
$\varepsilon$
$STATELIST \rightarrow sY$
$Y \rightarrow semi\ STATELIST$
$\varepsilon$

Чтобы показать, что данная грамматика относится к классу LL(1), достаточно рассмотреть множества первых порождаемых символов (DS) для двух продукций *X* и двух продукций *Y*. Для продукций, имеющих в левой части нетерминал *X*, имеем следующее.

$DS(X \rightarrow semi\ DECLIST) = \{semi\}$
$DS(X \rightarrow \varepsilon) = \{comma\}$

Последнее множество определено путем рассмотрения последователей *X*. Терминал *comma* является последователем *DECLIST*, что видно из следующего выражения.

$PROGRAM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$

В то же время любой последователь *DECLIST* является последователем *X*, что следует из продукции

$DECLIST \rightarrow dX$

Таким образом, *comma* является последователем *X*, а значит, первым порождаемым символом продукции

$X \rightarrow \varepsilon$

Подобным образом

$DS(Y \rightarrow semi\ STATELIST) = \{semi\}$
$DS(Y \rightarrow \varepsilon) = \{end\}$

поскольку *end* является последователем *STATELIST*, что видно из продукции

$PROGRAM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$

а любой последователь *STATELIST* является последователем Y, что следует из продукции

$STATELIST \rightarrow sY$

Таким образом, в каждом случае оба множества первых порождаемых символов являются непересекающимися, и грамматика относится к классу LL(1).

Ниже приводятся функции для каждого нетерминала грамматики: *PROGRAM*, *DECLIST*, *X*, *STATELIST*, *Y*. Код написан на языке C, но можно использовать любой другой язык, разрешающий применять рекурсивные функции.

```
void PROGRAM() /*соответствует PROGRAM*/
{ if (token! = begin)
  error();
  token = lexical();
  DECLIST();
  if (token! = comma)
  error();
  token = lexical();
  STATELIST();
  if (token! = end)
  error();
```

```
}

void DECLIST()
{ if (token! = d)
  error();
  token = lexical();
  X();
}

void X()
{ if (token == semi)
  { token = lexical();
    DECLIST();
  }
  else if (token == comma)
  ; /*ничего не делать*/
  else error();
}

void STATELIST()
{ if (token! = s)
  error();
  token = lexical();
  Y();
}

void Y()
{ if (token == semi)
  {   token = lexical();
      STATELIST();
  }
  else if (token == end)
  ; /*ничего не делать*/
  else error();
}

main ()
{ token = lexical();
  PROGRAM();
}
```

Вызов `lexical()` вынуждает лексический анализатор передать следующий символ синтаксическому анализатору, а `error()` вызывается при появлении синтаксической ошибки. Предполагается, что при вызове функции `error()` инициируются определенные процедуры восстановления после ошибок (здесь не конкретизируются). `semi`, `comma`, `begin` и `end` являются предварительно заданными константами, значениями которых являются представления данных символов этапа, следующего за лексическим анализом.

Порядок, в котором записаны функции, соответствует порядку продукций грамматики. Для компиляции посредством компилятора Borland C им должны

предшествовать следующие прототипы функций (это позволит применять функции до их объявления).

```
void DECLIST();
void STATELIST();
void X();
void Y();
```

В продукциях грамматики присутствует (хотя и неявно) рекурсия, следовательно, она имеется и в программе синтаксического анализа. Разумеется, если ни одна из продукций не содержит рекурсии, генерируемый язык будет крайне ограниченным и будет состоять только из конечного числа предложений. В то же время рекурсия в программе синтаксического анализа может быть дорогой, и ее можно избежать следующим образом. Продукции грамматики следует переписать с использованием расширенной формы записи, которая включает знак * с его обычным значением (нуль или более вхождений предшествующего элемента). Затем продукции могут записываться следующим образом.

*PROGRAM → begin DECLIST comma STATELIST end*
*DECLIST → d (semi d)**
*STATELIST → s (semi s)**

Данное представление грамматики компактнее и, пожалуй, читабельнее, чем приводившееся ранее. Использованная форма записи иногда называется расширенной формой Бэкуса-Наура, исходная форма записи эквивалентна форме Бэкуса-Наура, изначально использованной для определения языка ALGOL 60.

Программа синтаксического анализа, созданная на основе приведенных продукций, будет использовать уже не рекурсию, а итерации. Функции `main()` и `PROGRAM()` аналогичны приведенным выше, а функции `DECLIST()` и `STATELIST()` можно переписать следующим образом. Функции `X` и `Y` не нужны.

```
void DECLIST()
{ if (token! = d)
  error();
  token = lexical();
  while (token == semi)
  {token = lexical();
  if (token! = d)
  error ();
  token = lexical();
  }
}

void STATELIST()
{ if (token! = s)
  error();
  token = lexical();
  while (token == semi)
  {token = lexical();
```

```
    if (token! = s)
    error ();
    token = lexical();
    }
}
```

Здесь `lexical()`, `error()`, `semi`, `comma`, `begin` и `end` имеют то же значение, что и ранее.

С помощью описанного способа правую рекурсию всегда можно превратить в итерацию; кроме того, данный процесс можно автоматизировать. Левая рекурсия не может появляться в LL(1)-грамматике и, как было показано выше, ее можно преобразовать в правую рекурсию. Таким образом, правую и левую рекурсии можно рассматривать как итеративные, а не рекурсивные процедуры, поэтому программу синтаксического анализа создать возможно всегда.

В то же время грамматики для языка 2-го (но не 3-го) типа будут содержать среднюю рекурсию (например, для сопоставления с шаблоном), и это нельзя (точнее, нельзя легким способом) заменить итерацией. Рассмотрим, например, грамматику для выражений со следующими продукциями, в которых терминалы заключены в кавычки, чтобы не возникало путаницы между терминалами “*”, “(” и “)” и теми же знаками, использованными как метасимволы.

$E \rightarrow E$“+”$T$
$E \rightarrow T$
$T \rightarrow T$“*”$F$
$T \rightarrow F$
$F \rightarrow$ “(”$E$“)”
$F \rightarrow$ “$x$”

Данную грамматику можно преобразовать к виду LL(1).

$E \rightarrow TX$
$X \rightarrow$ “+”$TX$
$X \rightarrow \varepsilon$
$T \rightarrow FY$
$Y \rightarrow$ “*”$FY$
$Y \rightarrow \varepsilon$

$F \rightarrow$ “(”$E$“)”
$F \rightarrow$ “$x$”

Заменяя, где это возможно, рекурсию итерацией, получаем следующее.

$E \rightarrow T$(“+”$T$)*
$T \rightarrow F$(“*”$F$)*
$F \rightarrow$ “(”$E$“)”
$F \rightarrow$ “$x$”

Впрочем, остается (непрямая) средняя рекурсия, в которой фигурируют *E*, *T* и *F*, устранить которую нельзя. Функции рекурсивного спуска соответствуют (что неудивительно) также продукциям, содержащим не только итерацию, но и рекурсию (см. продукции для *E* и *T*). Следующая ниже реализация основана на приведенных выше продукциях, дополненных продукцией, которая вводит символ предложения, не появляющийся в правой части ни одной продукции.

$S \rightarrow E$
$E \rightarrow T(\text{“+”}T)*$
$T \rightarrow F(\text{“*”}F)*$
$F \rightarrow \text{“(”}E\text{“)”}$
$F \rightarrow \text{“}x\text{”}$

Функции имеют следующий вид.

```
void E()
{  T();
   while (token == plus)
   {token = lexical();
   T();
   }
}

void T()
{  F();
   while (token == times)
   {token = lexical();
   F();
   }
}

void F()
{  if (token == obracket)
   {token=lexical();
   E();
   if (token == cbracket)
   token=lexical();
   else error();
   }
   else if (token == x)
   token = lexical();
   else error();
}

main ()
{ token = lexical();
E();
}
```

Здесь `plus`, `times`, `obracket`, `cbracket` и `x` — представления на этапе, следующем за анализом, символов +, *, (, ) и *x* соответственно. Как и ранее, для компиляции функции в начале кода должны находиться прототипы для `E`, `F`, `T`.

Одним способом реализации эффекта средней рекурсии является введение в программу синтаксического анализа явного стека, который можно использовать для хранения адресов возврата для входа и выхода функции. Данный подход, похоже, эффективнее, чем более общий механизм обработки рекурсии, предлагаемый высокоуровневыми языками программирования.

# 4.5. Преобразования грамматик

Одним из основных ограничений анализа методом рекурсивного спуска, как и других методов LL(1)-анализа, является необходимость преобразования грамматики. При этом применяются два типа преобразования.

1. Удаление левой рекурсии.
2. Факторизация.

## 4.5.1. Удаление левой рекурсии

Левую рекурсию, как показывалось ранее, всегда можно удалить из контекстно-свободной грамматики. В то же время этот процесс следует проводить аккуратно, поскольку при этом изменяются значения строк, генерируемых изменяемыми продукциями. Например, требуется преобразовать левые рекурсии следующих продукций в правые.

$$E \rightarrow E + T$$
$$E \rightarrow T$$

Может показаться, что данные продукции следует изменить таким образом.

$$E \rightarrow T + E$$
$$E \rightarrow T$$

При этом генерируемые строки затронуты не будут. Действительно, это может быть все, что требуется. В то же время, если значение строк важно (например, при создании компилятора или нахождении значения выражения), то приведенная выше леворекурсивная форма генерирует предложение

$$T + T + T + T$$

следующим образом.

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow E + T + T + T \Rightarrow T + T + T + T$$

В данном случае подразумевается вычисление выражения слева направо, что ниже представлено посредством скобок.

$$((((T + T) + T) + T) + T)$$

В то же время праворекурсивная форма генерирует выражение следующим образом.

$$E \Rightarrow T + E \Rightarrow T + T + E \Rightarrow T + T + T + E \Rightarrow T + T + T + T$$

Здесь имеет место вычисление выражения справа налево.

$$(T + (T + (T + (T + T))))$$

Влияет ли порядок вычисления выражения на его значение, зависит от значения оператора +; для арифметического оператора + (по крайней мере, для целых чисел) порядок вычисления значения не имеет. Впрочем, компилятор обычно определяет конкретный порядок вычисления арифметических выражений, который, скорее всего, является простейшим и легчайшим в реализации. Считается, что в большинстве случаев проще реализовать вычисление слева направо, так что из приведенных выше продукций предпочтительнее леворекурсивные (по крайней мере, с указанной точки зрения).

Пожалуй, стоит сказать, что в тех случаях, когда невозможно подвести вычисление выражения слева направо под приведенные выше рекурсивные правила, реализовывать это будет неудобно и неестественно, поэтому следует что-либо изменить, дабы избежать такой ситуации.

Что действительно требуется — так это праворекурсивная грамматика, подразумевающая вычисление слева направо, и вот здесь находит применение использованное ранее преобразование, поскольку правила

$$E \to TX$$
$$X \to + TX$$
$$X \to \varepsilon$$

являются праворекурсивными и предполагают вычисление слева направо. Пусть генерируется следующее выражение.

$$T + T + T + T$$

Тогда порождение

$$E \Rightarrow T X \Rightarrow T + T X \Rightarrow T + T + T X \Rightarrow T + T + T + T X \Rightarrow T + T + T + T$$

предполагает следующую расстановку скобок.

$$((((T + T) + T) + T) + T)$$

К сожалению, преобразование продукций

$$E \to E + T$$
$$E \to T$$

в продукции

$$E \to TX$$
$$X \to + TX$$
$$X \to \varepsilon$$

может показаться не слишком естественным. Очевидно, что оно не настолько просто, как обращение порядка символов в первой продукции. В то же время

уже отмечалось, что данное преобразование не будет сложным, если рассматривать его с точки зрения *языка*, генерируемого правилами грамматики. В общем случае правила

$P \to Pa$
$P \to b$

генерируют язык

$ba^*$

Данные правила также могут генерироваться следующими продукциями.

$P \to bX$
$X \to aX$
$\varepsilon$

Полученный результат легко обобщить. Пусть имеется множество леворекурсивных продукций для нетерминала $P$ и множество продукций для $P$, которые не являются леворекурсивными.

$P \to P\alpha_1$, $P \to P\alpha_2$, $P \to P\alpha_3$, …, $P \to P\alpha_n$
$P \to \beta_1$, $P \to \beta_2$, $P \to \beta_3$, …, $P \to \beta_m$

Здесь символы $\beta$ не содержат $P$. Данные продукции генерируют следующее.

$(\beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_n)^*$

Данный язык можно также сгенерировать следующими группами продукций.

$P \to \beta_1$, $P \to \beta_2$, $P \to \beta_3$, …, $P \to \beta_m$
$P \to \beta_1 Z$, $P \to \beta_2 Z$, $P \to \beta_3 Z$, …, $P \to \beta_m Z$
$Z \to \alpha_1$, $Z \to \alpha_2$, $Z \to \alpha_3$, …, $Z \to \alpha_n$
$Z \to \alpha_1 Z$, $Z \to \alpha_2 Z$, $Z \to \alpha_3 Z$, …, $Z \to \alpha_n Z$

Здесь $Z$ — новый нетерминал, а продукции стали праворекурсивными. Следует отметить, что более общим, чем рассмотренный, случаем является непрямая рекурсия. Алгоритм устранения непрямой левой рекурсии заключается в первоначальной замене непрямой левой рекурсии прямой левой рекурсией (это можно сделать всегда, но соответствующий алгоритм мы приводить не будем), после чего задача сводится к рассмотренной выше.

Основным в приведенном выше обсуждении является наличие алгоритма удаления левой рекурсии из грамматики и замены ее правой рекурсией и то, что данный процесс можно автоматизировать, т.е. поручить программе. Использование программы делает процесс более надежным, менее подверженным человеческим ошибкам. Разумеется, не следует забывать о возможности совершить ошибку в самой программе преобразования, но частое использование программы должно подтвердить ее правильность.

## 4.5.2. Факторизация

Перейдем ко второму типу преобразований, которому должны подвергаться продукции грамматики для превращения ее в LL(1). Как говорилось выше, это преоб-

разование называется факторизацией, а его иллюстрация приводится ниже. Рассмотрим грамматику со следующими продукциями.

$P \to aPb$
$P \to aPc$
$P \to d$

Очевидно, что данная грамматика не является LL(1), поскольку две первых продукции в качестве первого порождаемого символа имеют $a$. Проблема устраняется путем преобразования данных продукций в такие.

$P \to aPX$
$X \to b$
$X \to c$
$P \to d$

Здесь было факторизовано $aP$, и появился новый нетерминал $X$. В качестве другого примера рассмотрим продукции

$P \to abQ$
$P \to acR$

Их можно преобразовать в следующие продукции.

$P \to aX$
$X \to bQ$
$X \to cR$

Процесс выглядит достаточно простым, так что может показаться, что произвести его можно всегда и что существует алгоритм преобразования грамматики, требующей факторизации, в LL(1)-грамматику. Впрочем, из следующего примера видно, что это не так. Рассмотрим продукции

$P \to Qx$
$P \to Ry$
$Q \to sQm$
$Q \to q$
$R \to sRn$
$R \to r$

Данная грамматика не является LL(1), поскольку первые две продукции в качестве первого порождаемого символа имеют $s$. Перед тем, как станет возможной факторизация, нетерминалы $Q$ и $R$ в первых двух продукциях нужно заменить, используя последние четыре продукции. Таким образом, вместо первых двух продукций получаем следующее.

$P \to sQmx$
$P \to qx$
$P \to sRny$
$P \to ry$

Полученные продукции можно факторизовать, что дает такой результат.

$P \rightarrow sP_1$
$P \rightarrow qx$
$P \rightarrow ry$

Здесь

$P_1 \rightarrow Qmx$
$P_1 \rightarrow Rny$

Перечислим все полученные продукции.

$P \rightarrow sP_1$
$P \rightarrow qx$
$P \rightarrow ry$
$P_1 \rightarrow Qmx$
$P_1 \rightarrow Rny$
$Q \rightarrow sQm$
$Q \rightarrow q$
$R \rightarrow sRn$
$R \rightarrow r$

Данные продукции по-прежнему не дают LL(1)-грамматики, поскольку обе продукции для $P_1$ в качестве первого порождаемого символа имеют $s$. Проблема идентична первоначальной, так что можем попытаться продолжить аналогичным образом. Для замены нетерминалов $Q$ и $R$ будут использованы четыре последние продукции, результат будет факторизован введением новой переменной $P_2$ — и все это только для того, чтобы обнаружить в точности ту же проблему с $P_2$, которая ранее была связана с $P$ и $P_1$. Процесс не прекратится никогда, но грамматика будет становиться все больше и больше. В разделе 4.3 уже отмечалось, что не существует алгоритма преобразования любой грамматики для LL(1)-языка в форму LL(1). Таким образом, неудивительно, что факторизация возможна не всегда, *даже* если язык является LL(1). Кстати, то, что алгоритм зацикливается и не дает LL(1)-грамматики, еще ничего не говорит о языке — является он LL(1) или нет.

Язык, генерируемый приведенной выше грамматикой, *не* является LL(1) и его можно выразить в следующем виде.

$\{s^i qm^i x \mid s^i rn^i y\}$

Данный язык нельзя проанализировать нисходящим образом, слева направо, поскольку при прочтении символа $s$ невозможно определить, появится далее такое же число символов $m$ или символов $n$, не используя неопределенное число символов предпросмотра, чтобы обнаружить следующий за $s$ символ $q$ или $r$.

Приведенный выше пример был искусственным, подобные свойства маловероятны в реальных языках программирования. Например, в языках программирования при наличии открывающей скобки обычно имеется единственный символ, представляющий соответствующую закрывающую скобку. Большин-

ство языков программирования (точнее, контекстно-свободные их аспекты) относятся к классу LL(1), следовательно, поддаются анализу методом рекурсивного спуска. Проблема заключается в том, что грамматики, используемые для представления языков программирования, не относятся к классу LL(1), так что перед разработкой программ синтаксического анализа методом рекурсивного спуска обычно необходимы преобразования грамматики. В следующем разделе этот вопрос рассматривается полнее, также обсуждаются другие достоинства и недостатки синтаксического анализа LL(1).

## 4.6. Достоинства и недостатки LL(1)-анализа

Причина привлекательности синтаксического анализа LL(1) заключается в его естественности, данный метод является крайне наглядным и удобным для создания основы для последующей компиляции языка программирования. Кроме того, его легко реализовать и убедиться в корректности его работы. Помимо выполнения собственно синтаксического анализа написанный код может содержать функции по выполнению *проверки соответствия типов* и других проверок, а также действия этапа синтеза, такие как распределения памяти и генерация кода.

В то же время можно определить и некоторые недостатки синтаксического анализа методом рекурсивного спуска, такие как неэффективность вызовов функций и необходимость преобразования грамматики, даже не зная, существует ли подходящее преобразование. Проблема заключается не только в нахождении преобразования, но и в проверке корректности его применения. Таким образом, имеются веские причины использовать при преобразовании надежные инструментальные средства, а не зависеть от ручного подхода. Среди других недостатков синтаксического анализа методом рекурсивного спуска можно выделить следующие.

- Часто создаются очень большие программы синтаксического анализа.
- Существует тенденция к появлению в теле одной функции операций, относящихся к разным фазам процесса компиляции.

К сожалению, последняя особенность не сильно улучшает общую структуру компилятора.

Для эффективного использования рекурсивного спуска требуется следующее.

- Хороший преобразователь грамматики, который *в большинстве случаев* сможет трансформировать грамматику в форму LL(1) — ранее показывалось, что, по теоретическим причинам, преобразователь не сможет выполнить свою работу для *всех* возможных входов.
- Возможность представить эквивалент программы синтаксического анализа методом рекурсивного спуска в табличной форме. Это означает, что при проверке входного текста программа будет не входить в функции и покидать их, а просто перемещаться по табличному эквиваленту грамматики, при необходимости занося в стек адреса возврата.

Хорошие преобразователи существуют и временами объединяются с инструментальными средствами и дают "таблицы" LL(1). Кроме того, те же инструменты могут позволять пользователям определять (причем в *исходной* грамматике) операции, которые следует выполнять на определенных этапах синтаксического анализа. Существенным преимуществом является задание операций именно относительно исходной, а не преобразованной, грамматики, поскольку пользователю удобнее мыслить понятиями исходной, более естественной грамматики, чем понятиями менее естественной LL(1)-грамматики, порожденной преобразователем. В частности, если в преобразованной грамматике будет отсутствовать левая рекурсия, она может быть в исходной грамматике, обеспечивая, таким образом, более естественную основу для определения операций времени компиляции, таких как генерация кода для вычисления выражений слева направо. На практике неестественная природа преобразованной грамматики никоим образом не должна существенно мешать создателю компилятора. В следующем разделе показывается, как в грамматике можно определить операции по выполнению действий во время компиляции.

## 4.7. Введение действий в грамматику

Классический способ анализа арифметических (и других) выражений перед генерацией машинного кода заключается в формировании постфиксной записи. В качестве примера постфиксной (иногда называемой обратной польской (reverse Polish)) формы записи рассмотрим следующее (инфиксное) выражение.

$(a + b) * (c + d)$

В постфиксной форме данное выражение имеет следующий вид.

$ab + cd + *$

Отметим, что при такой форме записи отсутствуют скобки и понятие приоритета оператора. Кроме того, если постфиксное выражение вычисляется слева направо, операнды каждого оператора известны до появления оператора. Эта особенность постфиксной формы записи делает ее относительно простой для создания выходного кода.

Рассмотрим грамматику, имеющую следующие продукции.

$S \rightarrow EXP$
$EXP \rightarrow TERM$
$EXP \rightarrow EXP + TERM$
$EXP \rightarrow EXP - TERM$
$TERM \rightarrow FACT$
$TERM \rightarrow TERM * FACT$
$TERM \rightarrow TERM / FACT$
$FACT \rightarrow -FACT$
$FACT \rightarrow VAR$

*FACT* → (*EXP*)
*VAR* → *a* | *b* | *c* | *d* | *e*

В число выражений, генерируемых данной грамматикой, входят следующие.

(*a* + *b*)∗*c*
*a*∗*b* + *c*
*a*∗*b* + *c*∗*d*∗*e*

Далее будем предполагать существование среды, в которой действия, введенные в грамматику, выполняются каждый раз, когда соответствующей частью грамматики генерируется код анализа. Для генерации постфиксных выражений в грамматику необходимо ввести три действия, которые обозначим через *A1*, *A2* и *A3*.

*S* → *EXP*
*EXP* → *TERM*
*EXP* → *EXP* + <*A1*>*TERM*<*A2*>
*EXP* → *EXP* – <*A1*>*TERM*<*A2*>
*TERM* → *FACT*
*TERM* → *TERM*∗ <*A1*>*FACT*<*A2*>
*TERM* → *TERM*/<*A1*>*FACT*<*A2*>
*FACT* → – <*A1*>*FACT*<*A2*>
*FACT* → *VAR*<*A3*>
*FACT* → (*EXP*)
*VAR* → *a* | *b* | *c* | *d* | *e*

Здесь для обособления действий были использованы угловые скобки.

Все операторы нужно занести в стек (действие <A1>) в том порядке, в котором их можно напечатать в соответствующее время (действие <A2>). С другой стороны, переменные (VAR) только читаются и печатаются (действие <A3>). Иные действия *отсутствуют*. Стек можно определить и инициализировать следующим образом.

```
char stack [3];
int ptr = 0;
```

Кроме того, определяется переменная, значение которой равно последнему символу.

```
char in;
```

В результате получаем такие три действия.

```
<A1>
{  stack[++ptr] = in;
}
   <A2>
{  printf("%c", stack [ptr--]);
}
<A3>
```

```
{   printf("%c", in);
}
```

Действия кажутся удивительно простыми, и, если быть честными, ситуация была несколько упрощена, поскольку по использованному определению переменные состоят только из одного символа. На первый взгляд, удивляет еще одно — действия не учитывают различные уровни приоритетов возможных операторов. Впрочем, понятие приоритета считается внедренным в исходную грамматику, так что нет необходимости что-либо знать о приоритете операторов или об использовании скобок. Видим, что действия, связанные с продукциями, содержащими скобки, отсутствуют.

Рассмотрим в качестве примера, как действия преобразуют следующее выражение грамматики.

(–*a* + *b*)* (*c* + *d*)

Чтобы показать использование различных действий, полезно продемонстрировать эффект генерации приведенного выше выражения грамматикой, содержащей действия.

(–<*A1*>*a*<*A3*><*A2*> + <*A1*>*b*<*A3*><*A2*>)* <*A1*>(*c*<*A3*> + <*A1*>*d*<*A3*><*A2*>)<*A2*>

Здесь показано, как при чтении строки в предложение вводятся действия. Результат действий можно представить в виде таблицы (табл. 4.2).

***Таблица 4.2.***

| *Считываемый символ* | *Действие* | *Содержимое стека* | *Выход* |
|---|---|---|---|
| ( | | | |
| – (минус) | A1 | – (минус) | |
| a | A3 | – | a |
| | A2 | – | – (минус) |
| + | A1 | + | |
| b | A3 | + | b |
| | A2 | + | + |
| ) | | | |
| * | A2 | * | |
| ( | | * | |
| c | A3 | * | c |
| + | A1 | *+ | |
| d | A3 | *+ | d |
| | A2 | * | + |
| ) | A2 | * | * |

Полный выход представляет прочтение последнего столбца сверху вниз, а верх стека предполагается находящимся справа. Достаточный размер стека — три элемента, поскольку существует всего три *различных* уровня приоритета

операторов (унарный, аддитивный и мультипликативный). Большее число уровней приоритета операторов потребует большего стека. Кроме того, при стеке произвольного размера может потребоваться правая рекурсия!

Разумеется, алгоритм зависит от доступности средств генерации синтаксических анализаторов, позволяющих создать код для чтения входа и соответствующего выполнения действий. Впрочем, такие средства имеются, и они предлагают мощное средство написания программ синтаксического анализа для чтения и выполнения действий над любым входом, который можно представить посредством контекстно-свободной грамматики. Типичным примером такого входа является исходный код; операции, которые можно произвести над этим кодом, разнообразны и их насчитывается множество — генерация выходного кода, использование перекрестных ссылок, проведение различных измерений и т.д. Мы пытались показать, что многие операции, обычно производимые над исходным кодом, простым и естественным образом выражаются как действия в контекстно-свободной грамматике. Выразив их именно таким образом и используя подходящие инструментальные средства, можно значительно упростить создание компиляторов и связанных с ними инструментальных средств. Становится не только просто писать компиляторы, облегчается их понимание, а значит, компиляторы проще модифицировать, а также проще отследить, насколько корректно они работают.

В следующих главах подробнее рассматриваются методы восходящего синтаксического анализа, а также связанные с этим процессом инструментальные средства. В процессе рассмотрения станут понятнее выгоды использования грамматики как основы действий времени компиляции.

## 4.8. Резюме

Данная глава посвящена нисходящему синтаксическому анализу. В частности, в ней было сделано следующее.

- Определены LL(1)-грамматики и языки.
- Показано, как на основе LL(1)-грамматик могут создаваться программы синтаксического анализа методом рекурсивного спуска.
- Показано, как определенные грамматики можно привести к виду LL(1), используя удаление левой рекурсии и факторизацию.
- Определены принципиальные преимущества и недостатки анализа методом рекурсивного спуска.
- Показано, как в программу синтаксического анализа можно ввести действия времени компиляции.

## Дополнительная литература

Терминология грамматик LL(1) вводится в книге [Knuth, 1971], а свойства таких грамматик — в книге [Foster, 1968]. Идея компиляция методом рекурсивного спуска уходит корнями в 1960-е, и ее авторство приписывается Лукасу [Lucas, 1961]. Она широко использовалась в 1970-х для создания переносимых компиляторов Pascal [Wirth, 1971]; пример компилятора для языка Pascal приводится в работе [Welsh and Hay, 1986].

Существует множество работ по компиляторам, созданных на основе метода рекурсивного спуска, и среди них можно выделить [Ullmann, 1994]. Инструментальные средства синтаксического анализа методом рекурсивного спуска описаны в работе [Terry, 1997]. Алгоритм, дающий ответ на вопрос принадлежности грамматики к классу LL(1), а также создающий для грамматики таблицу синтаксического анализа, описывается в [Aho, Sethi and Ullman, 1985].

## Упражнения

**4.1.** Приведите пример грамматики, не относящейся к классу LL(1), но генерирующей LL(1)-язык.

**4.2.** Объясните следующие факты.

а) Неоднозначная грамматика не может быть LL(1).
б) Грамматика, содержащая левую рекурсию, не может быть LL(1).

**4.3.** Покажите, что следующие языки относятся к классу LL(1), определив для каждого случая LL(1)-грамматику.

а) $\{x^n a y^n \mid n > 0\}$
б) $\{x^n a y^n \cup x^n a z^n \mid n >= 0\}$
в) $\{x^n a y^n \cup z^n a y^n \mid n >= 0\}$

($\cup$ — оператор объединения множеств)

**4.4.** Обсудите относительные преимущества и недостатки левой рекурсии в грамматиках с точки зрения синтаксического анализа.

**4.5.** Рассмотрим два набора продукций.

а) *DECLIST* → *d semi DECLIST*
*DECLIST* → *d*
б) *DECLIST* → *DECLIST semi d*
*DECLIST* → *d*

и предложение

*d semi d semi d*

Для каждого набора продукций а) и б) определите, какой терминал *d* порождается второй продукцией.

*DECLIST* → *d*

**4.6.** Приведите LL(1)-грамматики для следующих языков.

а) $\{0^n a 1^{2n} \mid n \geq 0\}$

б) $\{\alpha \mid \alpha$ принадлежит $\{0, 1\}^*$ и не содержит двух последовательно идущих единиц$\}$

в) $\{\alpha \mid \alpha$ состоит из равного числа нулей и единиц$\}$

**4.7.** Определите, относится ли грамматика со следующими продукциями к классу LL(1). Ответ аргументируйте.

$S \rightarrow AB$
$S \rightarrow PQx$
$A \rightarrow xy$
$A \rightarrow m$
$B \rightarrow bC$
$C \rightarrow bC$
$C \rightarrow \varepsilon$
$P \rightarrow pP$
$P \rightarrow \varepsilon$
$Q \rightarrow qQ$
$Q \rightarrow \varepsilon$

Здесь $S$ — символ предложения.

**4.8.** Опишите язык, генерируемый грамматикой из упражнения 4.7.

**4.9.** Преобразуйте грамматику со следующими ниже продукциями в форму LL(1).

$S \rightarrow EXP$
$EXP \rightarrow TERM$
$EXP \rightarrow EXP + TERM$
$EXP \rightarrow EXP - TERM$
$TERM \rightarrow FACT$
$TERM \rightarrow TERM * FACT$
$TERM \rightarrow TERM/FACT$
$FACT \rightarrow - FACT$
$FACT \rightarrow (EXP)$
$FACT \rightarrow VAR$
$VAR \rightarrow a \mid b \mid c \mid d \mid e$

**4.10.** Покажите, где в преобразованной грамматике упражнения 4.9 появятся действия, определенные в разделе 4.7 для производства постфиксной записи.